Identifying Performance Bottlenecks on Modern Microarchitectures using an Adaptable Probe

Gorden Griem*, Leonid Oliker*, John Shalf*, and Katherine Yelick*+
*Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA, 94720
+Computer Science Division, University of California, 387 Soda Hall #1776, Berkeley, CA 94720
{ggriem, loliker, jshalf, kayelick}@lbl.gov

Abstract— **The gap between peak performance and delivered performance for scientific applications running on microprocessor-based systems has grown considerably in recent years. The inability to achieve the desired performance even on a single processor is often attributed to an inadequate memory system, but without identification or quantification of a specific bottleneck. In this work, we use an adaptable probe to isolate application characteristics that cause a significant drop in performance, giving application programmers and architects information about possible optimizations. The probe uses only four parameters to capture key characteristics of scientific workloads: working-set size, computational intensity, indirection, and irregularity. We present and compare the results of the probe on four 64-bit high-performance architectures: Itanium 2, Opteron, Power3, and Power4. While all four systems show the expected gap between peak and delivered performance when the most challenging settings of the probe are used, the magnitude and point of the performance drop varies across machines, revealing interesting differences between them.**

Keywords— adaptable probe, microbenchmark, sqmat, Itanium 2, Opteron, Power3, Power4, performance analysis, scientific computing

I. Introduction

There is a growing gap between the peak performance of microprocessor-based systems and the delivered performance for scientific computing applications. This gap has raised the importance of developing better benchmarking methods to improve performance understanding and prediction, while identifying hardware and application features that work well or poorly together. Benchmarks are typically designed with two competing interests in mind – capturing real application workloads and identifying specific architectural bottlenecks that inhibit the performance. Benchmarking suites like the NAS Parallel Benchmarks [12] and the Standard Performance Evaluation Corporation (SPEC) [1] emphasize the first goal of representing real applications, but they are typically too large to run on simulated architectures and are too complex to employ for identification of specific architectural bottlenecks. Likewise, the complexity of the benchmarks can often end up benchmarking the quality of the compiler's optimizations as much as it does the underlying hardware architecture. At the other extreme, microbenchmarks such as STREAM [11] are used to measure the performance of a specific feature of a given computer architecture. Such synthetic benchmarks are often easier to optimize so as to minimize the dependence on the maturity of the compiler technology. However, the simplicity and narrow focus of the microbenchmarks often makes it quite difficult to relate to any real application code. Indeed, it is rare that such probes offer any predictive value for the performance of full-fledged scientific application.

Benchmarks, often present a narrow view of a broad, multidimensional parameter space of machine characteristics. We differentiate a "probe" from a "microbenchmark" or synthetic benchmark on the basis that the benchmark typically offers a single-valued result in order to rank processor performance consecutively – a few points of reference in a multidimensional space. A probe, by contrast, is used to explore a continuous, multidimensional parameter space. The probe's parameterization helps the researcher uncover the peaks and valleys in a continuum of performance characteristics and explore the ambiguities of computer architectural comparisons that cannot be captured by a single-valued ranking.

In this paper, we introduce the *sqmat* probe [14], which attempts to bridge the gap between these competing requirements. It maintains the simplicity of a microbenchmark while offering four distinct parameters to capture different types of application workloads:

- Working-set size (parameter "N")
- Computational intensity (parameter "M")
- Indirection (parameter "I")
- Irregularity (parameter "S")

These parameters can be varied to capture the characteristics of a broad range of scientific applications. The working-set-size represents the number of registers needed by the innermost loop in the algorithm. The computational intensity offers a measure of the density of floating point operations in the algorithm, expressed as a ratio of the number of floating point operations performed per memory access. Indirection is a binary choice that represents accesses of the form a[b[i]] that are common in sparse matrix codes. The irregularity/sparseness parameter measures the amount of spatial locality in the index vector of an indirect access. The amount of irregularity directly impacts the efficiency of the memory subsystem with respect to spatial locality. These four parameters are sufficient to give us insight into a wide variety of real scientific algorithm implementations while still retaining the simplicity needed to relate the results to specific characteristics of the architecture.

	MHz	GFlop/s	% of peak
Itanium 2	900	3.543	98.4 %
Opteron	1800	3.187	88.5 %
Power 3	375	1.405	93.7 %
Power 4	1300	3.780	72.7 %

Table 1: Dense matrix-matrix multiply performance according to [13]

Sqmat is based on matrix multiplication and is therefore related to the Linpack benchmark and to linear algebra solvers in general. The Linpack benchmark is used to rank the machines of the Top500 supercomputer list, although the benchmark reflects only a narrow class of large dense linear algebra applications. The results for dense matrix-multiply according to [13] for the four architectures are shown in Table 1 and can be taken as a reference point for the Sqmat performance.

In contrast, by varying the parameters of Sqmat, one can capture the memory system behavior of a more diverse set of applications as shown in Table 2. With a high computational intensity (M), the benchmark matches the characteristics of dense linear solvers that can be tiled into matrix-matrix operations (the so-called "BLAS-3" operations). For example PARATEC [15] is a material science application that performs ab-initio quantum-mechanical total energy calculations using pseudopotentials and a plane wave basis set. This code relies on BLAS-3 libraries with direct memory address-

	M	N	CI orig:sqmat	S	% irreg
DAXPY	1	1	0.5:0.5	-	0%
DGEMM	1	4	3.5:3.5	-	0%
MADCAP [17]	2	4	7.5:7.0	-	0%
SPMV	1	4	3.5:3.5	1	100%

Table 2: Sqmat parameters M, N, and S can be used to mimic the CI and % of irregular accesses of this representative selection of algorithms.

ing, thus having a high computational intensity, little indirection, and low irregularity. However, not all dense linear algebra problems can be tiled in this manner; instead they are organized as dense matrix-vector (BLAS-2) or vector-vector (BLAS-1) operations, which typically have only one or two operations on each element. This behavior is captured in sqmat by reducing the computational intensity, possibly in combination with a reduced working set size (N).

Indirection, sometimes called scatter/gather style memory access, occurs in sparse linear algebra, particle methods, and in grid-based applications with irregular domain boundaries. Most of these applications have the additional problem that access to memory are not contiguous, placing additional stress on memory systems that rely on large cache lines to mask memory latency. The amount of irregularity varies enormously in practice. For example, sparse matrices that arise in Finite Element applications often contain small dense sub-blocks, which cause a string of consecutive indexes in an indirect access of a sparse matrix-vector product. Another example can be found in GTC -- a magnetic fusion code that solves the gyro-averaged Vlasov-Poisson (gyrokinetic) system of equations using the particle-in-cell (PIC) approach [16]. The PIC method scales O(N) and requires calculations of particle motion, therefore this application has relatively low computational intensity, array indirection, and high irregularity. In both cases, the stream of memory accesses may contain sequences of contiguous memory accesses broken up by random-access jumps.

In this paper, we describe the implementation of the sqmat probe and focus on how its four parameters enable us to evaluate the behavior of four microprocessors that are popular building blocks for current high performance machines. The processors are compared on a basis of the delivered percentage of peak performance rather than absolute performance so as to limit the bias inherent in comparisons between different generations of microprocessor implementations. We evaluate and compare the strengths and weaknesses of these architectures and show where the performance penalties occur, giving application developers valuable hints on where to optimize their codes. Future work will focus on correlating sqmat parameter sets across a spectrum of scientific applications with varying computational and memory access requirements.

II. Introduction to sqmat

In this section we introduce the *sqmat* benchmark, which squares *L* matrices of size *NxN*. Each matrix is squared *M* times, i.e., raised to the power M. By varying M, the ratio between memory transfers and computation can be changed. By varying L, the memory the matrices are stored in is varied. So for large L, only small parts of the matrices can be stored in cache. In this work, we only use values of L large enough so that the matrix-entry array is several times the biggest cache size. With high probability, this forces the first load of each matrix to come from memory rather than from cache – even after repeating the benchmark multiple times in an outer loop in order to ensure consistent timing. Therefore, this paper does not explore the L parameter space as it is merely a machine-dependent constant.

The matrix entries can be accessed in two ways: Directly or indirectly. In the direct case, matrices are continuously stored in memory in row-major order. In the indirect case, there is still one block of memory the values are saved in, but now the parameter *S* controls the number of entries stored contiguously in memory: S entries are stored contiguously, then the next entry is at a random position in memory, followed by S-1 entries fol-

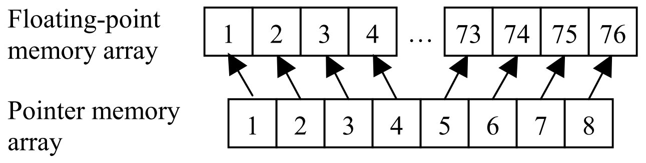

Figure 1: Example of Indirection for S=4. The numbers correspond to the relative position in the memory array.

lowing directly, and so on. Each entry owns a pointer pointing to the location of the value in memory space. Varying S varies the ratio of cached and non-cached memory accesses. Figure 1 shows an example of the memory layout for S=4.

Squaring the matrices is done in three steps: First, the values of one matrix are loaded from memory into registers. If there are more values than registers, register spills will occur. Next, the matrix is squared *M* times in the registers. Finally, the results are written back to memory. We use a Java program to generate optimally hand-unrolled C-code [14], greatly reducing the influence of the C-compiler's code generation and thereby making sure that the hardware architecture rather than the compiler is benchmarked. The unrolling is not so extreme as to cause processor stalls due to the increased number of instructions or additional branch-prediction penalties -- the innermost loop is only unrolled enough to ensure that all available floating-point registers are occupied by operands during each cycle of the loop. If enough registers are available on the target machine, several matrices are squared at the same time. Since squaring the matrix cannot be done in situ, an input and output matrix is needed for computation, thus a total of $2 \cdot N^2$ registers are required per matrix.

For direct access, each floating-point value has to be loaded and stored. As a double precision floating-point value is 8 bytes long [10], it creates 8 bytes memory-load and 8 bytes memory-store traffic for each value. For indirect access, the value and the pointer have to be

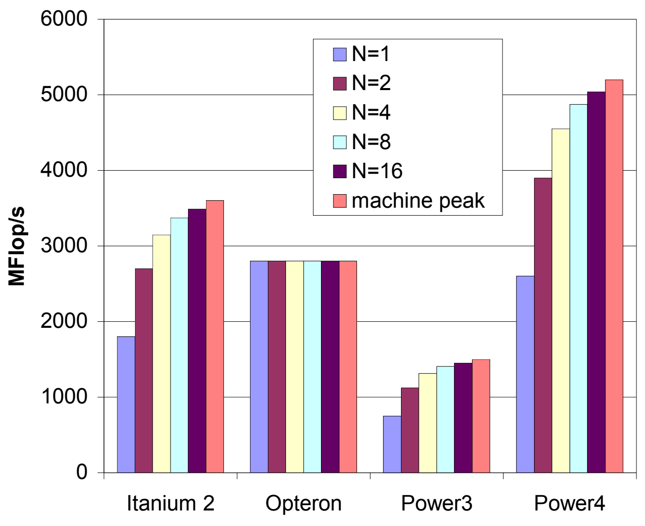

Figure 2: Algorithmic peak performance for different N

loaded. As we always use 64-bit pointers in 64-bit mode (avoiding legacy mode), each entry creates 16 bytes of memory-load and 8 bytes of memory-store traffic.

The sqmat benchmark is simple enough to be executed on existing hardware simulators, thereby allowing us to obtain performance estimates before the actual hardware is built. This topic will be explored in future work.

We have gathered performance results for the following four architectures: The Intel Itanium 2, the AMD Opteron, the IBM Power3, and the IBM Power4.

To allow a comparison between the different architectures, we introduce the *algorithmic peak performance (AP)*-metric. The AP is defined as the performance that could be achieved on the underlying hardware for a given algorithm if all the floating-point units were optimally used. The AP is always equal to or less than the *machine peak performance (MP)*. For example, some architectures support floating-point multiply-add instructions (FMA) and only achieve their peak rated flop rate when this instruction is used. Thus since a scalar multiply can only use the multiply-part of the instruction and not exploit the FMA, the effective maximum flop-rate is only half the MP for that processor. In this case, the AP would be significantly less than the MP. We therefore compare results against the AP, since FMA-capable architectures would be unreasonably punished when no FMA instructions are possible if the MP metric was used. Figure 2 shows the algorithmic peak performance for the four different architectures and different working-set sizes.

For each NxN matrix used by sqmat, updating an entry requires N multiplies and (N-1) adds. Therefore if the machine supports FMA (Itanium2, Power3, Power4), the AP is 1-1/(2·N) of the MP. For architectures not capable of executing FMAs (Opteron), the AP is equivalent to the MP.

To measure performance, we use the IBM HPMToolkit [3] on the Power3 and Power4 under AIX. For Itanium 2 and Opteron, we use the Performance Application Programming Interface (PAPI) [4] under Linux. Compilation is performed using the IBM xlc compiler on Power3 and Power4, the Portland Group's pgcc on the Opteron, and the Intel ecc on the Itanium 2.

All measurements were repeated thirty times and divided into ten runs. The normalized standard deviation between measurements was generally less than 1%. To make sure that operating-system dependent page-faults would not affect performance, we discard the first result of each run.

The remainder of this paper is organized as follows: Section III describes the hardware architectures, Section IV the performance for unit stride, Section V the effects of choosing different working set sizes, Section VI elaborates on indirection, Section VII shows the effects of irregularity, Section VIII the tolerance to irregularity, and in Section IX we draw overall conclusions.

III. Description of the Underlying Hardware

In this section, we describe the key components of the four processors in our study. These characteristics are shown in Table 3.

	Itanium 2	**Opteron**	**Power3**	**Power4**
Clock frequency	900 MHz	1.4 GHz	375 MHz	1.3 GHz
Number of FPU	2	2 (1 SSE)	2	2
FLOPS/cycle (non-FMA FLOPS/cycle)	4 (2)	2 (2)	4 (2)	4 (2)
FP Registers x nbits	128x64b	16x128b	32x64b	32x64b
Theoretical peak	3.6 GFlop/s	2.8 GFlop/s	1.5 GFlop/s	5.2 GFlop/s
Measured matmul peak [13]	3.5GF/s 98%peak	2.2GF/s 88%peak	1.4GF/s 94%peak	3.8GF/s 73%peak

Table 3: Floating point architectural characteristics of evaluated microprocessors.

A. Itanium 2:

The Intel Itanium 2 is a 64-bit processor with four floating-point units, two capable of executing one FMA per cycle while the other two perform other floating-point operations like comparisons. Only two floating-point operations can be executed in parallel [5]. Utilizing the maximum two FMAs per cycle using our test system running at 900 MHz, results in peak machine performance of 3.6 GFlop/s. The Itanium has 128 floating-point registers, so it is can accommodate matrices up to 8x8 (eg. N=8) in size and can pack several 4x4 matrices (N=4) into registers without spilling.

B. Opteron:

The primary floating point horsepower of the AMD Opteron comes from its SIMD floating-point unit accessed via the SSE2 instruction set extensions. The old x87 floating point unit and associated registers have been deprecated for all practical purposes. The Opteron can execute two double-precision floating-point operations per cycle using a single SIMD instruction on operands packed into 128-bit registers [6]. Therefore, the

1.4Ghz test system offers peak-performance of 2.8 GFlop/s. The 16 total 128-bit floating-point registers allow us to pack a single 4x4 matrix into registers without spilling (N=4), however the SIMD instructions require that two variables must be in the same 128-bit register to be operated on simultaneously. For a matrix-multiply, this cannot be guaranteed consistently, thus we expect the achieved performance to be significantly less than the algorithmic peak performance.

C. *Power3:*

The IBM Power3 processor has two floating-point units capable of executing one FMA per cycle [7]. Running at 375 MHz, this results in a peak-performance of 1.5 GFlop/s. The processor has 32 floating-point registers with an additional 32 rename registers that are not directly visible to the programmer; so one matrix of size 4x4 (N=4) fits into the registers. Despite its age and meager performance, the Power3 architecture is still widely used, most notably in systems that rank in the top ten slots of the Top500 supercomputer list [8].

D. *Power4:*

The IBM Power4 processor has two floating-point units capable of executing one FMA per cycle [9]. There are two processors on a die, but our experiments focus on only one processor. Running at 1.3 GHz, the peak performance is 5.2 GFlop/s. Of the 72 floating-point registers, only 32 are visible. The rest are rename registers for storing intermediate asynchronously generated results. One 4x4 matrix fits into the registers (N=4).

IV. COMPUTATIONAL INTENSITY WITH UNIT STRIDE:

In this section, we measure the performance of directly accessing the matrix entries on the four different architectures for N=4. By measuring the achieved performance for a given *computational intensity (CI)*, we evaluate how well the architecture hides the load/store latency to and from memory. We also measure the CI required to achieve 50% of algorithmic peak performance (lower is better). This indirectly measures the effective *throughput* of the processor's memory subsystem under a variety of conditions rather than its theoretical peak bandwidth. An architecture that is able to make effective use of its memory and cache subsystem will require less computational intensity to achieve 50% of algorithmic peak.

We define CI as the ratio of floating point operations to load and store operations. For architecture supporting FMA instructions, FMAs are counted as two separate operations. Given $M \cdot (2 \cdot N-1)$ operations per matrix entry two eight-byte memory operations for the direct access (three eight-byte memory operations for the indirect access) results in a CI of $M \cdot (2 \cdot N-1)/2$ for the direct case ($M \cdot (2 \cdot N-1)/3$ in the indirect case).

We investigate two potential performance bottlenecks in this section: For high M we expect performance to achieve AP. Otherwise, the functional units are not being used effectively. This is a symptom of an inability to find sufficient instruction level parallelism during instruction scheduling.

For small M, especially M=1, sqmat is essentially memory bound. Since all architectures are able to overlap computation with data transfers, we expect the bottleneck to be almost entirely due to data movement to and from registers. In Figure 3, the initial slope of the performance curves is directly proportional to the effective memory bandwidth while sqmat is still memory bound. As the CI reaches the point where it exceeds the effective bytes/flop ratio of the system, it levels off as sqmat performance becomes compute-bound – constrained by the throughput of the floating-point engine of the machine.

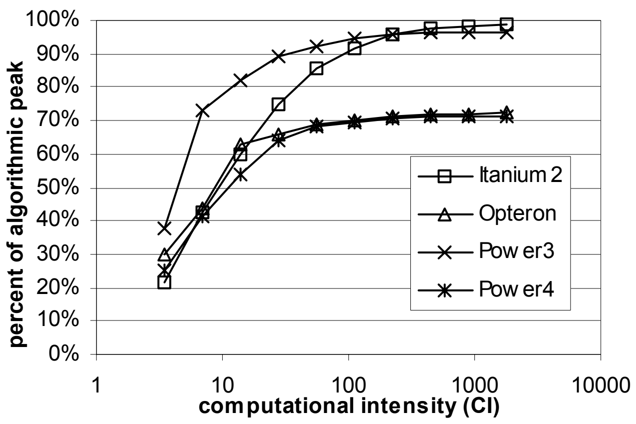

Figure 3: Achieved performance for unit stride, N=4

Figure 3 shows the achieved performance for different computational intensity for all four architectures using N=4 (note that CI=M·7/2). It can clearly be seen that Power3 and Itanium 2 are the only architectures achieving nearly algorithmic peak performance – the Opteron and Power4 trailing with 72.2% and 71.2% of AP respectively.

Whilst the performance is nearly stable for high CI on the other architectures, performance continues to improve on Itanium 2 as the CI grows, which together with the low performance for small M supports the assumption that floating-point transfers between cache and registers are a serious performance bottleneck on this platform. This may be attributed to its inability to store floating-point operands in the L1 cache.

Results also show the Power3 is effective at hiding the latency of cache access, while the Opteron and Power4 do not use all floating-point effectively. For the Opteron, this is due to the SIMD nature of the SSE instructions that require two floating point operations of the same type to be executed per cycle on operands in neighboring slots of its 128-bit registers – a constraint that cannot be satisfied at all times.

The Power4 does not have the same SIMD constraint as the Opteron, so to first order its superscalar execution unit was unable to find enough independent calculations to keep its two superscalar floating point execution units busy. Despite similarities to the Power3 superscalar functional units, the Power4 has much deeper execution pipelines (12 vs. 3 stages in the Power3) and thus putting more pressure on the instruction reordering to find instruction level parallelism (ILP). So we theorize that the increased pipeline depth is either inhibiting the Power4's ability to find sufficient ILP or is causing it to run short of the rename registers necessary to support the concurrent execution of the deeper pipelines of its floating point units

Machine	CI needed to achieve 50% of AP
Itanium 2	14
Opteron	14
Power 3	7
Power 4	14

Table 4: Computational intensity (measured in floating point operations per memory operation) needed to achieve 50% of algorithmic peak for N=4.

Table 4 shows the minimum computational intensity necessary to achieve 50% of algorithmic peak performance. The Power3 requires the lowest CI while the three more recent architectures (Itanium2, Opteron and Power4) need more computation to hide the data fetching.

V. Working Set Size:

In numerous applications, tiling is used to achieve better performance – the best-known example being dense matrix-multiply. In other applications, however, there are dependencies that prevent tiling or limit the tile size. In this section, we measure the effects of choosing different working-set sizes on the performance of the four processors.

In the sqmat benchmark, the parameter *N* controls the working-set size as it defines the size of the NxN matrix. For small N, each matrix will fit into registers. Larger N, however, will cause a register spill to L1 cache (L2 on

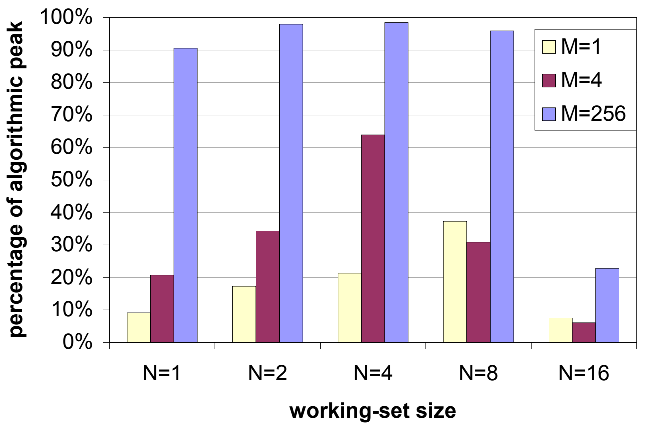

Figure 4: Achieved vs. Algorithmic Peak performance on the Intel Itanium 2

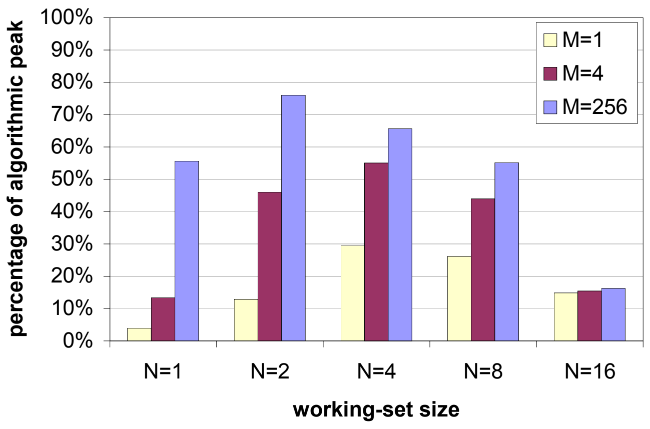

Figure 5: Achieved vs. Algorithmic peak performance on the AMD Opteron

Itanium). Since each element in the matrix has to be accessed N times, these register spills will have a dramatic impact on performance.

Performance is measured for N=1, 2, 4, 8, and 16, with an expectation that performance will drop when the working-set size exceeds the number of registers on a given platform. Note that for N=1, the matrix squaring degenerates to a scalar multiply.

We also investigate changes to the CI by varying M. For high M, algorithmic peak performance should be achieved when all elements are in registers. If register spills occur, we can measure the penalty by comparing performance with small N at high M. If additional registers are available, several matrices are squared concurrently, filling the pipeline with independent calculations. If these concurrent calculations were not executed, performance would suffer greatly. Most notably, even with the best result forwarding, only one calculation per cycle would be possible for N=1.

This part of the benchmark allows us to answer a number of questions about each machine. First, does a given machine ever reach the algorithmic peak? We will consider as many as 256 matrix squaring operations performed on in-register data, and expect ideal AP for such a high CI; otherwise instruction-scheduling problems are primarily at fault. Second, what is the optimal working-set size for each machine and does it correspond as expected to the number of available registers? Finally, what is the cost of register spilling when the working set size is too large to fit in registers? From the hardware designers' perspective, these results quantify the effects of the register set size and cache access latency. From

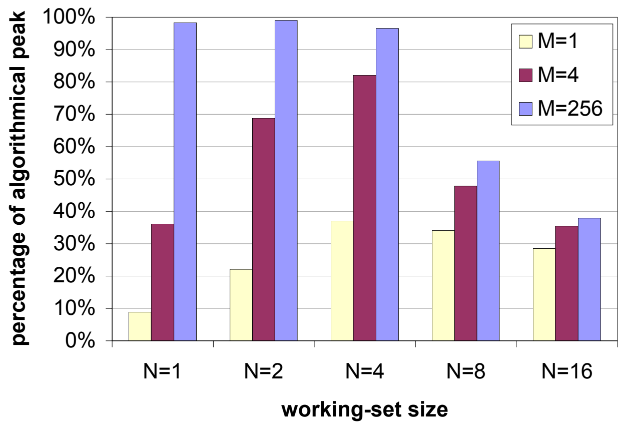

Figure 6: Achieved vs. Algorithmic peak performance for the IBM Power3

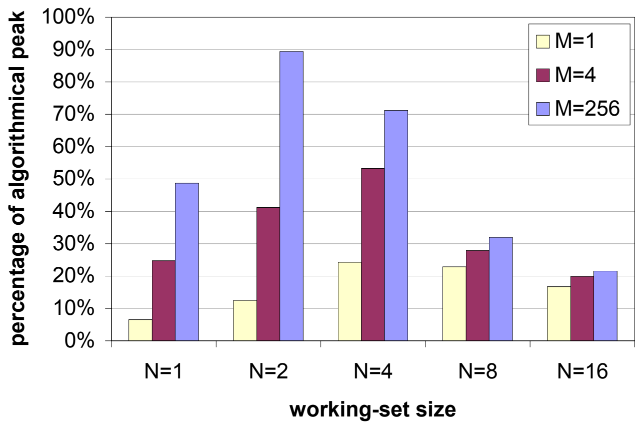

Figure 7: Achieved vs. Algorithmic peak performance for the IBM Power4

the software designers' perspective, they demonstrate the best working-set size, around which algorithms and compilers should be designed.

A. Itanium 2

The Itanium 2 processor cannot store floating-point variables in L1 cache but only in L2 cache; therefore, the impact of register spills and any failure of the compiler's instruction scheduling to effectively mask the latency of register loads is higher than for the other architectures.

Figure 4 shows the achieved performance for varying N and M. For small and medium N, algorithmic peak performance is achieved, showing that the Itanium 2 is capable of effectively using its hardware resources.

For high N, the penalty of register spills can clearly be seen. However, for high computational intensity, Itanium 2 is still capable of effectively hiding a certain number of register spills: N=8 achieves 95.9% for M=256). As a result, for high computational intensity, the working-set size can be chosen over a broader range than on the other evaluated architectures.

For N=16, only 22.9% of AP is achieved. This effect is less pronounced than expected given the 7 cycles penalty to access L2 cache suggests, but still shows that on Itanium 2, the working-set size has to be chosen carefully to avoid a performance penalty.

B. Opteron

With 16 128-bit and 8 MMX registers, the Opteron only has 40 floating-point double-precision registers. So for any N>4, register spills will occur. Furthermore, the probability of executing two useful instructions per cycle is rather low since the Opteron only allows one SIMD instruction per cycle, which is executed on one 128-bit register. As a result, achieved performance is significantly less than algorithmic peak.

Figure 5 shows the achieved vs. algorithmic peak performance on Opteron for different N and different M. The effects of the SIMD instructions can clearly be seen as even in the best case only 76% of AP is attained.

For the N=16 case (register spills), the performance is only 16% of AP. Therefore register spills cause a significant slowdown (a factor of approximately five) on the Opteron architecture.

C. Power 3

With 32 visible registers, register spills will occur on Power3 for N>4. Running at the low frequency of 375 MHz and having a bandwidth of 3.2 GB/s from L1 cache and 1.6 GB/s to L1 cache, while returning L1-cache-hits in one cycle, we expect the penalty for these spills to be small in comparison to the other architectures.

Figure 6 shows the achieved vs. algorithmic peak performance on the Power3 for varying N and M. For N<=4, nearly 100% of algorithmic peak performance is achieved for large CI. For bigger N, the effects of register spills can be seen, but are moderate even for the N=16 case, where 38% of peak performance is achieved. Power3 offers the most tolerant behavior of all architectures reviewed in this paper, due partially to its L1 cache that is accessible in one cycle.

D. Power4

With 32 visible floating-point registers, register spills will occur on Power4 for N>4. Having the same bandwidth of 3.2 GB/s to L1 cache and 1.6 GB/s from L1 cache as the Power3, but running at a higher frequency, we expect the penalty of these spills to be significantly more than for the Power3.

Figure 7 shows the achieved vs. algorithmic peak performance on the Power4 for varying N and M. Interestingly, the best performance is achieved for N=2 with

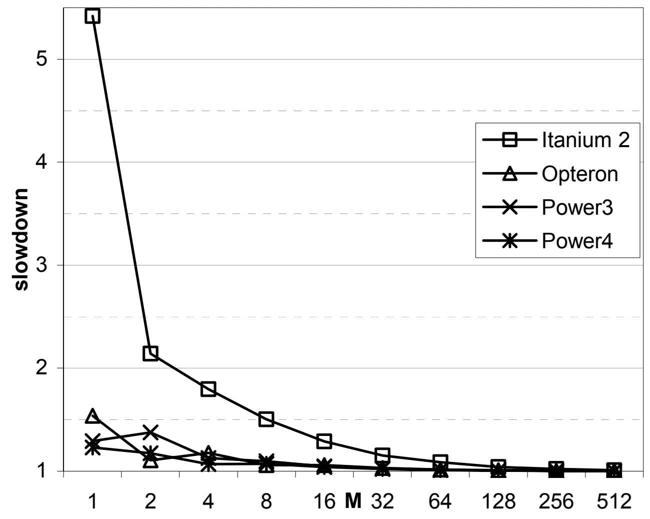

Figure 8: Slowdown due to Indirection relative to computational intensity for N=4. For direct access, the CI is M·7/2, for indirect access M·7/3

nearly 90% of AP. The N=4 case only achieves 72% of AP, which is even more surprising since we checked the generated assembly-code and found no visible register spills. We theorize that the deeper execution pipelines of the Power4 may exhaust rename register resources – thereby constraining the amount of instruction level parallelism. For N=16, only 22% of AP is achieved, showing that the penalty for register spills is significantly higher than on the Power3, but still better than on the other architectures.

VI. INDIRECTION:

In many applications, data cannot be accessed directly but has to be accessed indirectly via pointers. In this section, we measure the slowdown caused by adding one layer of indirection at a working-set size of N=4, thereby evaluating the third performance bottleneck, the effects of indirection.

We chose a model for indirection that mimics the compressed-row format of a typical sparse-matrix multiply implementation. The model we chose offers no direct correspondence to sparse algorithms that employ list or tree traversals; however, one can infer some more general conclusions by examining the effect of irregular access frequency on the peak throughput of these microprocessors.

In our indirect implementation, one continuous block of memory is allocated to store the floating-point values, while another contiguous block is allocated for the pointers. Each floating-point variable owns a pointer, with all data accesses to floating-point variables going through a level of indirection. Both pointers and values are stored in row-major order in memory; the matrices are stored continuously in memory.

The comparison between the direct and indirect implementation shows how well the architectures hide the overhead caused by indirect access through computation. It can be assumed that the memory-prefetch unit issues requests such that data is cache resident by the time it is needed. The slowdown can be attributed to two factors. As twice the memory load traffic is generated, the memory bandwidth may be too small to effectively fill the cache-lines before they are consumed. Additionally, there is instruction overhead caused by first getting the pointer and then the value. In theory, an architecture capable of rescheduling more instructions dynamically should be able to reduce the slowdown caused by the second factor as pointer calculations and floating-point arithmetic are executed in different functional units.

When comparing the direct and indirect case, we examine performance for a fixed M. However, the indirect case requires pointer access while the direct case does not, resulting in different CI for a given M. To circumvent this problem, we only use M for comparisons in this section, but keep in mind that the CI in the indirect case is only 2/3 of the CI of the direct case.

Figure 8 shows the slowdown for indirect access compared to the direct access for different CI. Opteron, Power3 and Power4 behave approximately the same, with the penalty being less than 10% for M>8. This demonstrates that the bandwidth between cache and processor is high enough to deliver both addresses and values. On Power3 and Power4, this can be explained by the fact that L1 to processor bandwidth is twice as much as processor to L1. It can also be seen that the longer rescheduling queue of Power4 achieves a better performance than the shorter queue of the Power3.

Finally, on the Itanium 2, indirection results in a significantly higher penalty even for high computational complexities that are very difficult to achieve in real applications: For M>1, the penalty is a factor of 5.4x, while even at M=8, the slowdown remains high at 1.5x. These results show that indirection is a significant bottleneck on Itanium 2 for reasons that we've thus far been unable to completely understand. We are currently investigating this issue.

VII. IRREGULARITY:

Numerous scientific applications, such as sparse matrix-matrix multiply, have certain irregular patterns in their data accesses. In general, solvers that performs arithmetic on spare matrices attempt to improve time-to-solution by skipping elements that are either zero, or otherwise too small to contribute meaningfully to the solution. The most typical sparse matrix representation is a row-compressed format where an index array is employed to skip over the zeros in the source matrix using indirect references of the form SourceVector[IndexArray[I]]. In practice, the access pattern encoded in the indirection array appears as a set of contiguous memory accesses, followed by a jump that skips over zero elements, then followed by another set of contiguous accesses, and so on. The size of the jumps is typically greater than the size of a cache line while the length of the contiguous accesses is entirely problem-dependent. We therefore introduce the parameter *S* to model these access patterns for problems that exhibit varying degrees of irregularity.

Sets of S floating-point values are stored contiguously in memory at random starting positions. Therefore when traversing the data linearly the first element is at a random position, the next S-1 elements are directly following, then the next element is at a random position, and so on. The pointers are pre-computed and stored contiguously in memory. The starting address for the S contiguous floating-point values is set to N·8·S to align the memory layout with the cache lines therefore minimizing unnecessarily splitting elements across multiple cache lines. For an example of the memory layout, refer to Figure 1. For all the shown measurements, we used L=100,000 which is mapped into an allocated memory space that is double that size, resulting in 25.6MB of memory usage for the floating-point variables and 12.8MB of memory reserved for pointer variables.

Assuming that the memory-prefetch unit is sufficiently intelligent, the first of S elements in any given contiguous block of non-zeros will most likely result in cache-misses. However, due to our cache-line oriented data alignment, the next S-1 elements will be fetched into cache appropriately. Therefore, by varying S, we can change the ratio of cache hits to cache misses. Note that performance results are compared with indirect memory access where all elements are stored contiguously (denoted as $S=\infty$).

When S is not a power of 2, we observed performance degradation for increasing S due to cache-line misalignment. Thus we restrict our experiments of S to powers

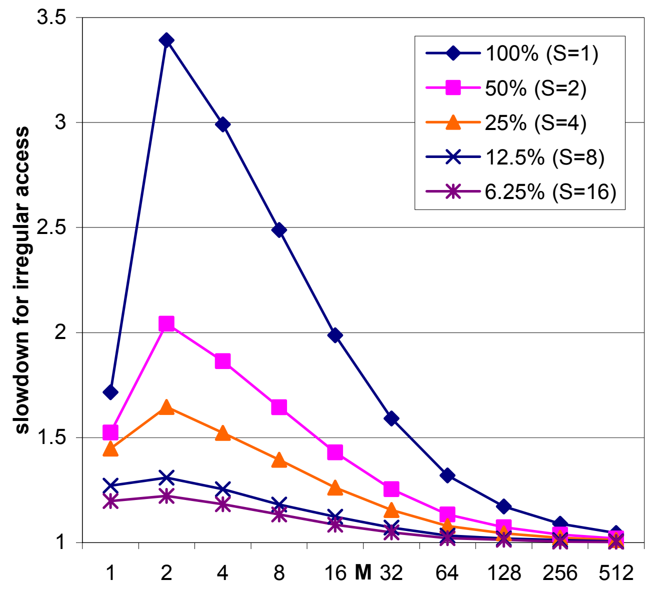

Figure 9: Irregularity on Itanium 2 for N=4

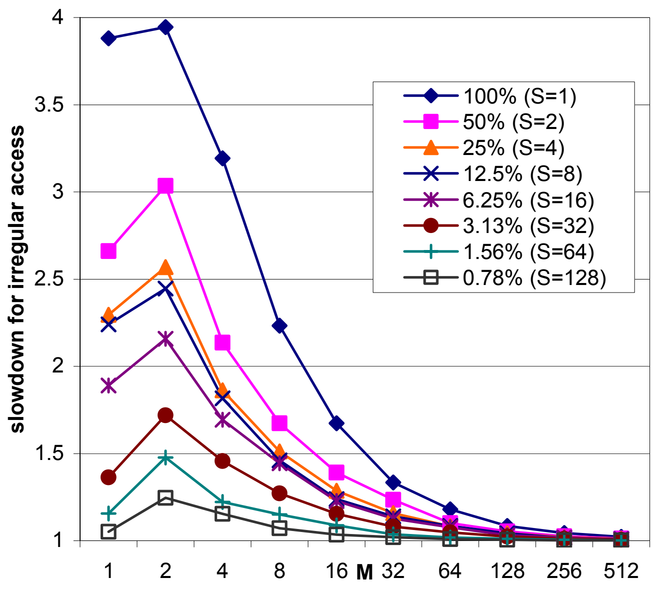

Figure 10: Irregularity on Opteron for N=4

of two. For all architectures, S=1…16 was chosen; however, on selected platforms where S>16 resulted in significantly different results, we show performance for high S (up to 256). By comparing the slowdown between decreasing S and S=∞, we measure the architecture's tolerance to cache-misses. This part of the benchmark primarily measures memory subsystem throughput. Bandwidth to memory, memory access latency and the number of outstanding cache misses an architecture can effectively handle are all key performance factors in this section.

A. Itanium 2

As Figure 9 shows, the Itanium 2 performs extremely well for irregular access. Even with S=1, the worst slowdown (at M=2) is only 3.39 times worse than regular access, making the Itanium 2 the best evaluated architecture for irregularity. For random accesses as high

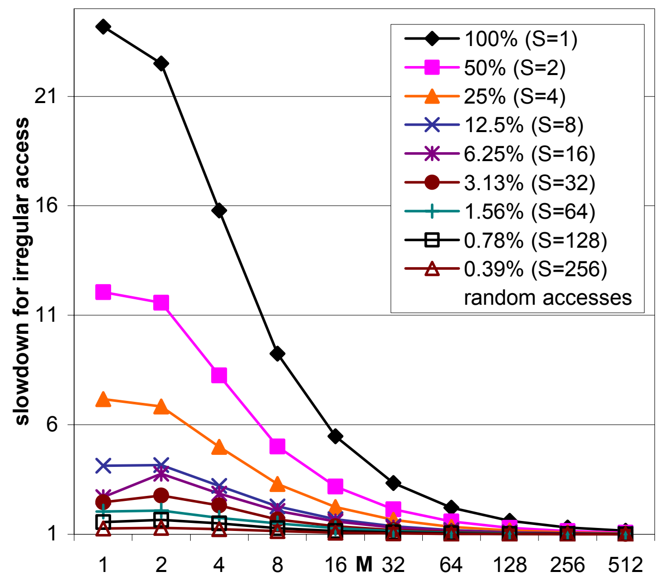

Figure 11: Irregularity on Power3 for N=4

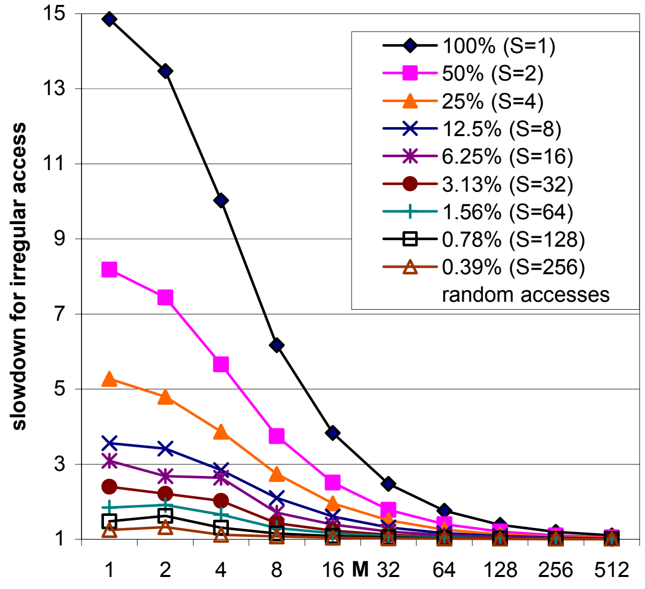

Figure 12: Irregularity on Power4 for N=4

as 12.5% (S=8), the performance degradation for irregular access is less than 31% even for smallest computational intensity.

These results can partly be explained by the fact that floating-point variables only get cached in the L2, making the cost of cache misses less expensive in comparison to other systems. It is also shown that the architecture and/or the compiler instruction scheduler is effective at hiding the penalties of cache misses in this situation.

B. *Opteron*

Opteron performs nearly as well as the Itanium 2 for totally random accesses, as seen in Figure 10. At M=4, the penalty for 100% random access (S=1) is only a factor of 3.19, compared to 2.99 on the Itanium 2. As the frequency of irregular access increases, however, performance improves at a much slower rate than for the Itanium 2. Even at S=128, when only a tiny fraction of the elements are out of unit stride (0.78%), the performance penalty is still up to 25%.

These results show that the shorter memory access latency enabled by the on-chip memory controller combined with the Opteron's ability to sustain up to eight outstanding cache misses, allows the memory system to perform well for irregular data accesses. However, the architecture is not as effective as the Itanium2 in hiding latency for small numbers of irregular accesses.

C. *Power3*

As seen in Figure 11, the Power3 performs poorly for irregular data access. For (M=1, S=1), total random accesses slow down performance by a factor of more than 24 compared to linear access; more than six times worse compared with the Opteron and Itanium2. Even an irregular access with 256 consecutive accesses in memory performs significantly slower than the S=∞ case.

The poor performance for S=1 can mostly be explained by the 35 cycle penalty of a cache-miss [8] and the fact that the Power3 memory subsystem does not allow more than four prefetch streams from memory to L1 cache.

It is also likely that the prefetch engines contribute to this slowdown. The prefetch engines require a long stream of contiguous accesses in order to detect a viable prefetch stream. Even a small amount of irregularity can confuse these relatively simple hardware units. When the prefetch engines cannot be engaged, the execution engine is subjected to the full round-trip memory latency as cache lines are loaded on an essentially demand-driven basis. The compiler can override this behavior with explicit prefetch directives, but there is insufficient information at compile time to make this choice.

Results show that the Power3's memory subsystem is the most intolerant of irregularity of the platforms we examined. The Power3 architecture was clearly optimized to handle dense-mode numerical kernels.

D. *Power4*

As can be seen in Figure 12, the cost of irregular access is high on the Power4. For (M=1, S-1), the penalty for all-random accesses is a factor of 14.8 compared to regular accesses (M=1, S=∞). Although performing better than the Power3, even S=256 is significantly worse than the S=∞ case.

Given the Power4's comparatively deep instruction reordering capability – able to manage 200 instructions in flight per cycle -- one would expect a performance comparable to the Opteron. However, such deep reordering is apparently insufficient to hide the memory latency for this level of memory access irregularity.

The 512-byte L3 cache line size partially contributes to the inefficient memory performance for highly irregular problems, but it is insufficient to fully explain the slowdown for the problems that exhibit a very small number of irregular accesses. It is likely that the hardware detection of prefetch streams is involved in this behavior. The Power4's prefetch units require four contiguous cache line references (64 double words) to ramp up to full speed, thereby avoiding many unnecessary fetches caused by false predictions. A single indirection will cancel that prefetch stream. Thus the hardware is "tricked" very easily with only slight irregularity. Future work will examine this issue in more detail.

The Power4 has a "data cache block touch" (dcbt) instruction that immediately engage a hardware prefetch stream without the ramp-up; however the compiler does not have sufficient information to automatically insert the dcbt instruction since its benefit depends on the degree of irregularity – a determination that can only be made at runtime. Aggressively inserting the dcbt instruction will hurt performance considerably more for cases that are highly irregular than it helps for cases that less so. This highlights the limitations of relying on compile-time analysis to make appropriate instruction scheduling decisions. The Power3 and Power4 have architectural features that were designed specifically to benefit dense mode numerical kernels, but these very same features have a deleterious effect on irregular/sparse mode algorithms. There is clearly a need to deliver architectural features that address the needs of scientific codes that have both dense and irregular access patterns.

VIII. BALANCE:

In this section, we use the results of Section VII to describe the architectural tolerance to irregularity by introducing two metrics: *M50* and *S50*.

We ask two questions in regard to architectural tolerance for irregularity: (a) how much irregularity can the architecture hide at a given computational intensity, and (b) how much computation is needed to hide the worst possible irregularity (i.e. accessing each entry at a random position).

The answer to (a) is given in the *S50* value: Let the performance of the indirect unit stride access ($S=\infty$) at a given computational intensity be P_∞. Let the smallest S achieving at least 50% be P_∞ (at the same CI) is the S50 value. Thus the S50 tells us how much irregularity can be tolerated by the architecture with a 50% performance loss.

Question (b) is answered by the *M50* value: Let the performance of the indirect unit stride access ($S=\infty$) at the lowest possible computational intensity, i.e. M=1 (CI=7/3), be P_1. The smallest M that achieves 50% of P_1 given all random accesses (S=1) is the M50 value. Thus the M50 value tells us how much computational intensity is needed to hide the irregular access to maintain 50% of performance.

Figure 13 shows the S50 values for computational intensity for M=1 ($CI=7/3\cong2.3$) and M=8 ($CI=56/3\cong18.7$). It can clearly be seen that Itanium 2 performs extremely well, tolerating 50% of random memory accesses at a CI of 18.7, and showing an architectural example where the gap between internal frequency and memory accesses is bridged rather successfully.

The Opteron also performs well -- tolerating 50% of random accesses at a CI of 18.7. However, at a low CI of 2.3, the S50 value is only 6.3%, performing significantly worse than the Itanium 2.

Power 4 only tolerates one out of 64 random accesses at a CI of 2.3 and still needs 16 consecutive elements for a CI of 18.7. This is far worse than both Itanium 2 and Opteron. These results are consistent with the Power4's prefetch stream policy that requires 64 consecutive word requests to engage a stream. For low computational intensity, more than 2% of random accesses result in a

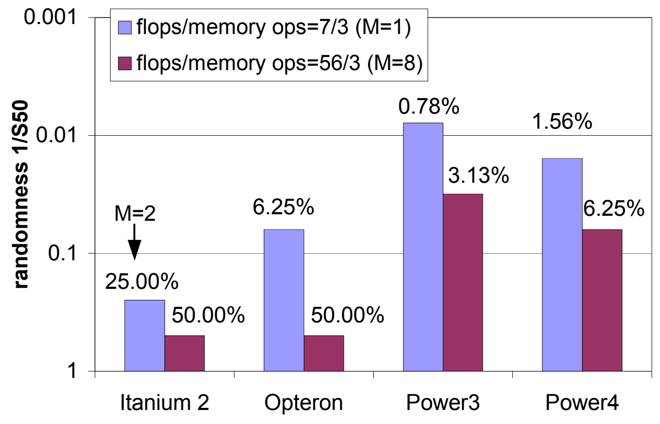

Figure 13: S50 values for different CI for N=4. The values shown are the randomness 1/S50.

50% performance penalty. Even for a relatively high CI, only 6.3% of random accesses are tolerated.

Power 3 shows the worst performance, with one out of 128 elements tolerated for M=1 and one out of 32 elements tolerated for M=8. In other words: if less than 1% of memory accesses are not consecutive, the penalty of performance on the Power3 can already be more than 50%! As irregularity of less than 1%, while relevant for codes employing dense-mode BLAS-3 cores, is unrealistic for newer scientific applications that are increasingly moving towards sparse data representations.

Figure 14 presents the M50 values. The figure shows a similar behavior as the S50 figure, with Itanium 2's M50=4 (CI=9.3), Opteron's M50=8 (CI=18.7), Power4's M50=32 (CI=74.7), and Power3's M50=64 (CI=149.3). Thus the Power3 needs a computational intensity of 150 to achieve a reasonable performance. It is worth noting that although the Power4 is running a much higher internal frequency than the Power3, and subsequently widening the gap between memory and internal frequency, it outperforms the Power3. However the Power4 still performs far worse than the Itanium 2 or Opteron. The Power3 and Power4's architectural enhancements to benefit dense-mode numerics hurt its sparse mode performance.

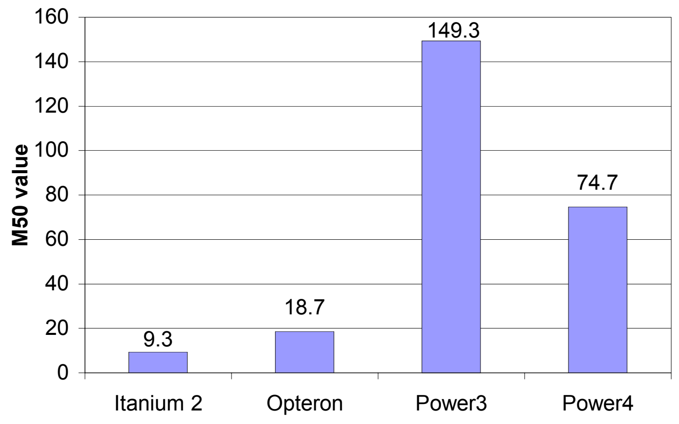

Figure 14: M50 values translated into CI for N=4

In conclusion, this section showed that the architecture of the Itanium 2 is most forgiving for random accesses, followed by the Opteron. The Power3 and Power4 need huge M50 and S50 values, demonstrating that they are more suitable for dense-mode algorithms. We believe that there is a large movement towards sparse representations as they emphasize time-to-solution over peak flop-rate. However, the computational intensity required to gain any advantage in moving to a sparse representation is dauntingly large for these microarchitectures. There is a clear need for architectural enhancements to improve efficiency of sparse access patterns in order to keep pace with the current state-of-the-art numerical solver design trends.

IX. Conclusions and Future Work

In this paper, we have introduced a new adjustable synthetic benchmark (a "probe") that measures the performance of a single processor and its memory system. With only a few parameters, the benchmark is capable of giving an estimate for the runtime of serial codes based on their working-set size, computational complexity, and the irregularity of their memory access pattern. The results are given relative to an algorithmic peak, which allows comparisons across different floating-point unit implementations as well as different generations of microprocessor architectures.

Probes like sqmat complement the capabilities of traditional benchmarks as tools that enhance understanding of performance behavior of scientific codes. The probe allows searches through a continuous parameter space of algorithm performance characteristics, thereby supporting analysis of system architectural balance and design trade-offs for microprocessors. It also points out trade-offs that can be made by programmers on these architectures, such as employing large CI to compensate for irregular memory access patterns -- perhaps necessitating a move from explicit to implicit solvers.

It is important to take all of these results in context. The results on processors with the highest clock-rates in this study indicate that programmers should expect the gap between hardware peak and observed performance to continue to increase; however the high clock rates may still indicate better time-to-solution than the other machines. The decreased efficiency of the Power4 versus the Power3 despite similar superscalar functional unit and instruction set architecture illustrates the effect of deeper execution pipelines and the increasing impact of memory latency on computational efficiency as clock frequencies climb. Even a deep instruction reorder pipeline cannot hide the latency incurred by a cache miss during irregular accesses. The Opteron demonstrates that lowering the memory latency using an on-board memory controller is an effective method to attack this problem. Likewise, the Itanium2 uses a large register set and deep explicit prefetch queues to hide this latency. Out-of-order instruction processing appears to have more limited effectiveness in addressing this problem. However, none of these implementation offer a sustainable path for improvement as processor core speeds continue to outstrip reductions in memory subsystem latency. There is a critical need for future microprocessors to add architectural enhancements for addressing the needs of applications exhibiting this kind of memory access irregularity.

Our future plans include running the benchmark on vector machines, especially the Cray X1, which are optimized for irregular memory access. We are also planning to map performance counter results to performance to investigate the correlation between hardware counters and achieved performance. Our long-term goal is to isolate the architectural bottlenecks that cause the performance drops by looking at the performance counter results. These new tools will greatly assist in evaluating system architectures optimized for scientific workloads as well as providing a better understanding of code performance on existing architectures.

X. Acknowledgements

The authors would like to thank Paul Hargrove, Eric Roman, Hongzhang Shan, and Parry Husbands for their thoughtful contributions. The authors were supported by Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

XI. REFERENCES:

[1] Standard Performance Evaluation Corporation, http://www.specbench.org

[2] Performance Evaluation Research Center, http://perc.nersc.gov

[3] HPM Tool Kit, http://www.alphaworks.ibm.com/tech/hpmtoolkit

[4] Performance Application Programming Interface, http://icl.cs.utk.edu/projects/papi/

[5] Intel Itanium 2 Processor Reference Manual, http://www.intel.com/design/itanium2/manuals/251110.htm

[6] AMD64 Architecture Programmer's Manual, http://www.developwithamd.com

[7] RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide, IBM Redbook, http://www.redbooks.ibm.com

[8] 21st Top500 Supercomputer Sites, http://www.top500.org

[9] The POWER4 Processor: Introduction and Tuning Guide, IBM Redbook, http://www.redbooks.ibm.com

[10] ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic

[11] STREAM: Sustainable Memory Bandwidth in High Performance Computers, J.D. McCalpin, Dept. of Computer Science, University of Delaware, http://www.streambench.org/

[12] NAS Parallel Benchmarks: http://science.nas.nasa.gov/Software/NPB

[13] Kazushige Goto and Robert van de Geijn. On Reducing TLB Misses in Matrix Multiplication. FLAME Working Note #9, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-55. Nov. 2002.

[14] http://dit.lbl.gov/Bench/sqmat

[15] M.C. Payne, M.P. Teter, D.C. Allan, T.A. Arias, and J.D. Joannopoulos, Iterative minimization techniques for ab initio total-energy calculations: Molecular dynamics and conjugate gradients, Rev. Mod. Phys., 64 (1993) 1045--1098.

[16] Z. Lin, S. Ethier, T.S. Hahm, and W.M. Tang. Size scaling of turbulent transport in magnetically confined plasmas. Phys. Rev. Lett., 88:195004, 2002.

[17] J. Borrill, MADCAP: The Microwave Anisotropy Dataset Computational Analysis Package, in: Proc. 5th European SGI/Cray MPP Workshop (Bologna, Italy, 1999) astro-ph/9911389.